Welcome to CS 536:
Introduction to Programming Languages and Compilers!

Instructor: Beck Hasti
- hasti@cs.wisc.edu
- Office hours to be determined

TAs
- Andrey Yao
- Robert Nagel
- Sadman Sakib
- Saikumar Yadugiri
- Ting Cai

Course websites:
- canvas.wisc.edu
- www.piazza.com/wisc/spring2024/compsci536
- pages.cs.wisc.edu/~hasti/cs536

About the course
We will study compilers
We will understand how they work
We will build a full compiler

Course mechanics
Exams (60%)
- Midterm 1 (18%): Thursday, February 29, 7:30 – 9 pm
- Midterm 2 (16%): Thursday, March 21, 7:30 – 9 pm
- Final (26%): Sunday, May 5, 2:45 – 4:45 pm

Programming Assignments (40%)
- 6 programs: 5% + 7% + 7% + 7% + 7% + 7%

Homework Assignments
- 8 short homeworks (optional, not graded)
What is a compiler?

A compiler is
- a recognizer of language S
- a translator from S to T
- a program in language H

Front end vs back end

front end = understand source code S; map S to IR

IR = intermediate representation

back end = map IR to T
Overview of typical compiler

Source program
  sequence of characters
  
  Scanner
  sequence of tokens
  
  Parser
  Syntax analyzer (Abstract Syntax Tree)
  name analysis
  type checking
  
  Semantic analyzer
  augmented, annotated AST
  
  Intermediate code generator
  IR
  
  Optimizer
  optimized IR
  
  Code generator
  assembly or machine code
  or target code

Symbol table

front end

back end
Scanner

Input: characters from source program
Output: sequence of tokens
Actions:
- group characters into lexemes (tokens)
- identify and ignore whitespace, comments, etc.
What errors can it catch?
- bad characters
  - `#` in Java
- unterminated strings
  - "Hello"
- integer literals that are too large

Parser

Input: sequence of tokens from the scanner
Output: AST (abstract syntax tree)
Actions:
- group tokens into sentences
What errors can it catch?
- syntax errors
  - `X = Y = 5;`
- (possibly) static semantic errors
  - use of undeclared variable

Semantic analyzer

Input: AST
Output: annotated AST
Actions: does more static semantic checks
- Name analysis
  - process decls & use of variables
  - match uses w/decls
  - enforces scoping rules
  - error - multiply-declared vars, use of undeclared variables
- Type checking
  - check types & augment AST

Intermediate code generator

Input: annotated AST
Output: intermediate representation (IR)
- assumes no syntax / static-semantic errors
  - `3-address code`
  - instructions have at most 3 operands
  - easy to generate from AST
  - 1 instr per AST internal node
Example

\[ a = 2 \times b + \text{abs}(-71); \]

Scanner produces tokens:

\text{ID(a) ASSIGN INTLIT(2) TIMES ID(b) PLUS ID(abs) LPAREN MINUS INTLIT(71) RPAREN SEMICOLON}

AST (from parser)

```
  ASSIGN
    ID
    PLUS
      MULT
        INTLIT
        ID
      INT
    INT
```

Symbol table Name analysis

gives us symbol table!

<table>
<thead>
<tr>
<th>ID</th>
<th>Kind</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>a, b</td>
<td>var</td>
<td>int</td>
</tr>
<tr>
<td>abs</td>
<td>func</td>
<td>int</td>
</tr>
</tbody>
</table>

3-address code

\begin{align*}
\text{temp1} &= 2 \times b \\
\text{temp2} &= 0 - 71 \\
\text{move temp2 param1} \\
\text{call abs} \\
\text{move return1 temp3} \\
\text{temp4} &= \text{temp1} + \text{temp3} \\
\text{a} &= \text{temp4}
\end{align*}
Optimizer

Input: IR
Output: optimized IR
Actions: improve code
- make it run faster, make it smaller
- several passes: local and global optimization
- more time spent in compilation; less time in execution

Code generator

Input: IR from optimizer
Output: target code

Symbol Table

Compiler keeps track of names in
- semantic analyzer - both name analysis & type checking
- code generation - off sets into stack
- optimizer - could we to keep track of def-use info

P1: implement symbol table

Block-structured language  
- nested visibility of names - no access outside of scope of name  
- easy to tell which def of a name applies  
- lifetime of data is bound to scope

Example: (from C)

```c
int x, y;

void A() {
    double x, z;
    C(x, y, z);
}

void B() {
    C(x, y, z);
}
```

block structure ⇒
- need nesting of sym tabs
⇒ list of hash-tables